Feasible reactivity in a synchronous pi-calculus

Roberto M. Amadio, Frederic Dabrowski

Feasible reactivity in a synchronous π-calculus*

Roberto M. Amadio
Université Paris 7 †

Frédéric Dabrowski
INRIA Sophia-Antipolis

February 11, 2007

Abstract

Reactivity is an essential property of a synchronous program. Informally, it guarantees that at each instant the program fed with an input will 'react' producing an output. In the present work, we consider a refined property that we call *feasible reactivity*. Beyond reactivity, this property guarantees that at each instant both the size of the program and its reaction time are bounded by a polynomial in the size of the parameters at the beginning of the computation and the size of the largest input. We propose a method to annotate programs and we develop related static analysis techniques that guarantee feasible reactivity for programs expressed in the $S\pi$-calculus. The latter is a synchronous version of the π-calculus based on the SL synchronous programming model.

1 Introduction

Mastering the *computational complexity* of programs is an important aspect of computer security with applications ranging from embedded systems to mobile code and smartcards. One approach to this problem is to monitor at run time the resource consumption and to rise an exception when some bound is reached. A variant of this approach is to instrument the code so that bounds are checked at appropriate time. An alternative approach is to analyse *statically* the program to guarantee that during the execution it will respect certain resource bounds. In other words, the first approach performs a *dynamic* verification while the second relies on a *static* analysis. As usual, the main advantage of the first approach is its flexibility while the advantage of the second approach is the fact that it does not introduce an overhead at run time and, perhaps more importantly, that it allows an early detection of 'buggy' programs. In this work, we will focus on the static analyses which offer the more challenging problems while keeping in mind that the two approaches are complementary. For instance, static analyses may be helpful in reducing the frequency of dynamic verifications.

*Work partially supported by ACI CRISS and ANR-06-SETI-010-02.
†Laboratoire *Preuves,Programmes et Systèmes*, UMR-CNRS 7126.

When addressing the issue of resource control, there is a variety of properties of a program that one may check. Termination is probably the first one that comes to mind. However, in the context of *interactive programs*, this property should be refined into *reactivity*. In general, the set of reactive programs can be defined (co-inductively) as the *largest* set R of programs that terminate and such that each interaction with the environment leads to a program which is again in the set R.

If a program manipulates data values of *variable size* such as lists, trees, graphs, ... then the analysis can go beyond reactivity and, for instance, it can establish that the program reacts while using a *feasible* amount of resources where feasible can be understood, for instance, as computable in *polynomial time*. In this case, the analysis produces a *function* that bounds the time (or space) needed for the reaction depending on the size of certain parameters.

There is a large collection of static analysis techniques (see, *e.g.*, [11, 7, 12, 13, 14, 9] that allow to establish feasible reactivity of *functional* programs. A common feature of these methods is the combination of traditional termination methods with what could be called a *data-size flow analysis*. By this we mean a method to describe how the size of the values computed by a program depends on the size of the values taken in input.

In [5, 6], we have started a research programme that aims at extending this approach to a synchronous, concurrent programming language. In the present work, we focus in particular on the $S\pi$-calculus [2]. This is a *synchronous* version of the π-calculus [18] which is based on the SL (synchronous language) model [10]. The latter can be regarded as a relaxation of the ESTEREL model [8] where the reaction to the *absence* of a signal within an instant can only happen at the next instant. Various full fledged concurrent and synchronous programming languages have been developed on top of the SL model (see, *e.g.*, [19, 15]) and the $S\pi$-calculus can be regarded as a more refined model capturing some essential aspects of those languages.

Our contribution includes (i) a methodology to annotate programs and (ii) related static analysis methods that guarantee feasible reactivity for *finite control* programs expressed in the $S\pi$-calculus.

Programs come with two kinds of *annotations* that concern *thread identifiers* and *signals*. A characteristic of synchronous programs is that each thread performs some set of actions in a *cyclic way*. A cycle is different from an instant in that it can span several instants (possibly an unbounded number of them). We require that a subset of the thread identifiers mark the end of a cycle and the beginning of a new one. This annotation has no effect on the operational semantics but it is used to produce certain static conditions. The first condition is what we call the *read once condition*. Informally, this condition requires that each thread within each cycle can only read a finite number of signals. The technical consequence of this restriction is that the behaviour of a thread within an instant can be described as a *function* of its parameters and the (finitely many) values read within the same cycle.

Thread identifiers carry two additional annotations. A basic goal is to show that each instant terminates. We are then naturally lead to *compare* thread identifiers and their parameters according to some suitable well-founded order. For this reason we assume

that each thread identifier is annotated with a *status* that describes how its parameters should be compared (typically, according to a lexicographic or multi-set order). Another important goal towards feasible reactivity, is to show that the parameters of a thread are in a sense *non-size increasing*. It turns out that it is not always appropriate to consider *all* parameters and therefore we require that we explicitly associate with each thread identifier the (possibly proper) subset of parameters that should be considered in the analysis of its size. To summarise, a thread identifier has three kinds of annotations: one saying whether it marks the end of a cycle, another, that we call status, describing how its parameters have to compared for termination analysis, and a final one specifying the subset of the parameters that are relevant to the computation of its size.

On one hand, a program should be allowed to emit values on a signal that depend on values read on other signals. On the other hand, we want to avoid situations where, for example, a program repeatedly reads a value on a signal and emits a larger value on the same signal. To address this issue, we assume that signal names are partitioned into a finite number of *regions* which are ordered. More precisely, we refine the type system so that signal types come with a region ρ as in the type $Sig_\rho(t)$. In other terms, the type of a signal name explicitly carries the information on the region to which the signal name belongs. Again, this annotation does not affect the operational semantics but it is used in the generation of static conditions that guarantee feasible reactivity. Informally, the condition states that the size of a value emitted on a signal at region ρ is bound by a function of the size of the values read from signals of smaller regions.

Next, we move on to an informal description of the static conditions. First of all, we have to find an abstract way to describe the data-size flow of a program. To this end, we import and adapt the concept of *quasi-interpretation* that has been proposed in the context of the analysis of the computational complexity of *first-order functional programs* [9, 4]. As a second step, we describe a method to associate with a program a finite set of inequalities on first-order terms and prove that whenever these inequalities are satisfied by a (polynomially bounded) quasi-interpretation the program is feasibly reactive. The inequalities can be classified in three categories according to their purpose which is to ensure: (1) the termination of each instant, (2) that the size of the parameters of a thread at the beginning of each cycle is non-size increasing, (3) that the size of the values computed by a thread within a cycle is bounded by a polynomial in the size of the parameters of the thread and the size of the values read on the signals within the cycle. Obviously, these inequalities depend on the signal and thread annotations we described above.

The rest of the paper is organised as follows. In section 2 we introduce the syntax of the $S\pi$-calculus along with some programming examples and an informal comparison with the π-calculus. In section 3, we provide the formal reduction semantics of the $S\pi$-calculus and we introduce the notion of feasible reactivity. In section 4, we define the different kinds of thread and signal annotations mentioned above, we show how to associate a set of inequalities with an annotated program, and we introduce the notion of assignment which provides an interpretation of the inequalities in terms of numerical functions. A quasi-interpretation is a polynomially bounded assignment which satisfies the inequalities. Our main result states that a program that admits a quasi-interpretation is feasibly reactive.

We devote section 5 to an outline of the proof techniques leaving the details in an appendix.

2 The $S\pi$-calculus

We introduce the syntax of the $S\pi$-calculus along with some programming examples and an informal comparison with the π-calculus.

2.1 Programs

Programs $P, Q, \ldots$ in the $S\pi$-calculus are defined as follows:

$$\begin{array}{lll} P & ::= & 0 \mid A(\mathbf{e}) \mid \overline{s}e \mid s(x).P, K \mid [s_1 = s_2]\ P_1, P_2 \mid [u \trianglerighteq p]\ P_1, P_2 \mid \nu s\ P \mid P_1 \mid P_2 \\ K & ::= & A(\mathbf{r}) \end{array}$$

We use the notation $\mathbf{m}$ for a vector $m_1, \ldots, m_n$, $n \geq 0$. The informal behaviour of programs follows. 0 is the terminated thread. $A(\mathbf{e})$ is a (tail) recursive call of a thread identifier A with a vector $\mathbf{e}$ of expressions as argument; as usual the thread identifier A is defined by a unique equation $A(\mathbf{x}) = P$ such that the free variables of P occur in $\mathbf{x}$. $\overline{s}e$ evaluates the expression e and emits its value on the signal s. $s(x).P, K$ is the *present* statement which is the fundamental operator of the SL model. If the values $v_1, \ldots, v_n$ have been emitted on the signal s then $s(x).P, K$ evolves non-deterministically into $[v_i/x]P$ for some v_i ($[\_/\_]$ is our notation for substitution). On the other hand, if no value is emitted then the continuation K is evaluated at the end of the instant. $[s_1 = s_2]\ P_1, P_2$ is the usual matching function of the π-calculus that runs P_1 if $s_1 = s_2$ and P_2, otherwise. Here both s_1 and s_2 are free. $[u \trianglerighteq p]\ P_1, P_2$, matches u against the pattern p. We assume u is either a variable x or a value v and p has the shape $\mathsf{c}(\mathbf{p})$, where c is a constructor and $\mathbf{p}$ is a vector of patterns. We also assume that if u is a variable x then x does not occur free in P_1. At run time, u is always a *value* and we run σP_1 if σ is the substitution matching u against p if it exists, and P_2 otherwise. Note that as usual the variables occurring in the pattern p (including signal names) are bound. $\nu s\ P$ creates a new signal name s and runs P. $(P_1 \mid P_2)$ runs in parallel P_1 and P_2. The continuation K is simply a recursive call whose arguments are either expressions or values associated with signals at the end of the instant in a sense that we explain below. We will also write $\mathsf{pause}.K$ for $\nu s\ s(x).0, K$ with s not free in K. This is the program that waits till the end of the instant and then evaluates K.

2.2 Expressions

The definition of programs relies on the following syntactic categories:

Sig	$::= s \mid t \mid \cdots$	(signal names)
Var	$::= Sig \mid x \mid y \mid z \mid \cdots$	(variables)
$Cnst$	$::= * \mid \mathsf{nil} \mid \mathsf{cons} \mid \mathsf{c} \mid \mathsf{d} \mid \cdots$	(constructors)
Val	$::= Sig \mid Cnst(Val, \ldots, Val)$	(values $v, v', \ldots$)
Pat	$::= Var \mid Cnst(Pat, \ldots, Pat)$	(patterns $p, p', \ldots$)
Fun	$::= f \mid g \mid \cdots$	(first-order function symbols)
Exp	$::= Var \mid Cnst(Exp, \ldots, Exp) \mid Fun(Exp, \ldots, Exp)$	(expressions $e, e', \ldots$)
$Rexp$	$::= !Sig \mid Var \mid Cnst(Rexp, \ldots, Rexp) \mid$ $Fun(Rexp, \ldots, Rexp)$	(exp. with deref. $r, r', \ldots$)

As in the π-calculus, signal names stand both for signal constants as generated by the ν operator and signal variables as in the formal parameter of the present operator. Variables Var include signal names as well as variables of other types. Constructors $Cnst$ include $*$, nil, and cons. Values Val are terms built out of constructors and signal names. The *size of a value* $|v|$ is defined as $|s| = |\mathsf{c}| = 0$ if c is a constant, and $|\mathsf{c}(v_1, \ldots, v_n)| = 1 + \Sigma_{i=1,\ldots,n}|v_i|$ if $n \geq 1$. Patterns Pat are terms built out of constructors and variables (including signal names). We assume first-order function symbols $f, g, \ldots$ whose behaviour will be defined axiomatically. Expressions Exp are terms built out of variables, constructors, and function symbols. Finally, $Rexp$ are expressions that may include the value associated with a signal s at the end of the instant (which is written $!s$, following the ML notation for dereferenciation). Intuitively, this value is a *list of values* representing the set of values emitted on the signal during the instant. If P, p are a program and a pattern then we denote with $fn(P), fn(p)$ the set of free signal names occurring in them, respectively. We also use $FV(P), FV(p)$ to denote the set of free variables (including signal names).

2.3 Typing

Types include the basic type 1 inhabited by the constant $*$ and, assuming t is a type, the type $Sig(t)$ of signals carrying values of type t, and the type $list(t)$ of lists of values of type t with constructors nil and cons. In the examples, it will be convenient to abbreviate $\mathsf{cons}(v_1, \ldots, \mathsf{cons}(v_n, \mathsf{nil}) \ldots)$ with $[v_1; \ldots; v_n]$. 1 and $list(t)$ are examples of *inductive types*. More inductive types (booleans, numbers, trees,...) can be added along with more constructors. We assume that variables (including signals), constructor symbols, and thread identifiers come with their (first-order) types. For instance, a constructor c may have a type $(t_1, t_2) \to t$ meaning that it waits two arguments of type t_1 and t_2 respectively and returns a value of type t. It is then straightforward to define when a program is well-typed and verify that this property is preserved by the following reduction semantics. We just notice that if a signal name s has type $Sig(t)$ then its dereferenced value $!s$ should have type $list(t)$. In the following, we will tacitly assume that we are handling well typed programs, expressions, substitutions,...

2.4 Comparison with the π-calculus

The syntax of the $S\pi$-calculus is similar to the one of the π-calculus, however there are some important *semantic* differences to keep in mind.

Deadlock vs. End of instant. What happens when all threads are either terminated or waiting for an event that cannot occur? In the π-calculus, the computation stops. In the $S\pi$-calculus (and more generally, in the SL model), this situation is detected and marks the end of the current instant. Then suspended threads are reinitialised, signals are reset, and the computation moves to the following instant.

Channels vs. Signals. In the π-calculus, a message is consumed by its recipient. In the $S\pi$-calculus, a value emitted along a signal persists within an instant and it is reset at the end of it. We note that in the semantics the only relevant information is whether a given value was emitted or not, *e.g.*, we do not distinguish the situation where the same value is emitted once or twice within an instant.

Data types. The (polyadic) π-calculus has *tuples* as basic data type, while the $S\pi$-calculus has *lists*. The reason for including lists rather than tuples in the *basic* calculus is that at the end of the instant we transform a set of values into a suitable data structure (in our case a list) that represents the set and that can be processed as a whole in the following instant. Note in particular, that the list associated with a signal is nil if and only if no value was emitted on the signal during the instant. This allows to detect the *absence* of a signal at the end of the instant.

We consider a simple example that illustrates our discussion. Assume $v_1 \neq v_2$ are two distinct values and consider the following program in $S\pi$:

$$P = \nu\ s_1, s_2\ (\quad \overline{s_1}v_1 \quad | \quad \overline{s_1}v_2 \quad | \quad s_1(x).\ (s_1(y).\ (s_2(z).\ A(x,y)\ \underline{,B(!s_1)}) \quad \underline{,0}) \quad \underline{,0} \quad)$$

If we forget about the underlined parts and we regard s_1, s_2 as *channel names* then P could also be viewed as a π-calculus process. In this case, P would reduce to

$$P_1 = \nu s_1, s_2\ (s_2(z).A(\sigma(x), \sigma(y))$$

where σ is a substitution such that $\sigma(x), \sigma(y) \in \{v_1, v_2\}$ and $\sigma(x) \neq \sigma(y)$. In $S\pi$, *signals persist within the instant* and P reduces to

$$P_2 = \nu s_1, s_2\ (\overline{s_1}v_1 \mid \overline{s_1}v_2 \mid (s_2(z).A(\sigma(x), \sigma(y)), B(!s_1)))$$

where $\sigma(x), \sigma(y) \in \{v_1, v_2\}$. What happens next? In the π-calculus, P_1 is *deadlocked* and no further computation is possible. In the $S\pi$-calculus, the fact that no further computation is possible in P_2 is detected and marks the *end of the current instant*. Then an additional computation represented by the relation $\mapsto$ moves P_2 to the following instant:

$$P_2 \mapsto P_2' = \nu s_1, s_2\ B(v)$$

where $v \in \{[v_1; v_2], [v_2; v_1]\}$. Thus at the end of the instant, a dereferenced signal such as $!s_1$ becomes a list of (distinct) values emitted on s_1 during the instant and then all signals are reset.

2.5 Programming examples

We introduce a few programming examples on which we will rely in the following to illustrate our static analysis techniques.

Example 1. *The synchronous model is particularly adapted to the* simulation *of various kinds of systems (we refer to [17] for a number of examples). Here, we describe the behaviour of a cell of a generic cellular automaton. Each cell relies on three parameters: its own activation signal s, its state q, and the list ℓ of activation signals of its neighbours. The cell performs the following operations in a cyclic fashion: (i) it emits its current state on the activation signals of its neighbours, (ii) it suspends for the current instant, and (iii) it collects the values emitted by its neighbours and computes its new state. This behaviour can be programmed as follows:*

$$\begin{array}{lcll} Cell(s,q,\ell) & = & Send(s,q,\ell,\ell) & \\ Send(s,q,\ell,\ell') & = & [\ell' \trianglerighteq \mathsf{cons}(s',\ell'')] & (\overline{s'}q \mid Send(s,q,\ell,\ell'')), \\ & & & \mathsf{pause}.Cell(s,next(q,!s),\ell) \end{array}$$

where next is a function that computes the following state of the cell according to its current state and the state of its neighbours. We assume some finite enumerated type 'state' that contains a constant for each state. The type of the signals s, s' is $Sig(state)$, the type of the lists ℓ, ℓ' is $list(Sig(state))$, and the type of the function next is $state, list(state) \to state$.

Example 2. *This example describes a 'server' handling a list of requests emitted in the previous instant on the signal s. For each request of the shape $\mathsf{req}(s', x)$, it provides an answer which is a function of x along the signal s'.*

$$\begin{array}{lcl} Server(s) & = & \mathsf{pause}.Handle(s,!s) \\ Handle(s,\ell) & = & [l \trianglerighteq \mathsf{cons}(\mathsf{req}(s',x),\ell')]\ (\overline{s'}f(x) \mid Handle(s,\ell')), Server(s) \end{array}$$

Assume the function f has type $t \to t'$ and assume an inductive type treq with a constructor $\mathsf{req} : Sig(t'), t \to treq$. Then the parameters s have type $Sig(treq)$ and the lists ℓ, ℓ' have type $list(treq)$.

Example 3. *This example describes two threads: the thread $A(s)$ re-emits on s the values that were emitted on s in the previous instant while the thread $C(s)$ emits a (fresh) value on s.*

$$\begin{array}{ll} A(s) & = \mathsf{pause}.B(s,!s) \\ B(s,\ell) & = [\ell \trianglerighteq \mathsf{cons}(n,\ell')]\ (\overline{s}n \mid B(s,\ell')), A(s) \\ C(s) & = \nu n\ \overline{s}n \mid \mathsf{pause}.C(s) \end{array}$$

Assuming n has type $Sig(1)$, s has type $Sig(Sig(1))$, and the list ℓ has type $list(Sig(1))$.

3 Reduction semantics and feasible reactivity

We provide the formal reduction semantics of the $S\pi$-calculus and we introduce the notion of feasible reactivity.

3.1 Expression evaluation

We assume an evaluation relation $\Downarrow$ such that for every function symbol f and values $v_1, \ldots, v_n$ of suitable type there is a unique value v such that $f(v_1, \ldots, v_n) \Downarrow v$, $fn(v) \subseteq \bigcup_{i=1,\ldots,n} fn(v_i)$, and moreover we suppose that the value v can be computed in time polynomial in the size of the values $v_1, \ldots, v_n$. As already mentioned, the techniques for defining first-order functional programs that enjoy these properties are well-studied. The evaluation relation $\Downarrow$ is extended to expressions as usual:

$$\frac{}{s \Downarrow s} \qquad \frac{e_i \Downarrow v_i \quad i = 1, \ldots, n}{\mathsf{c}(e_1, \ldots, e_n) \Downarrow \mathsf{c}(v_1, \ldots, v_n)} \qquad \frac{e_i \Downarrow v_i \quad i = 1, \ldots, n \quad f(v_1, \ldots, v_n) \Downarrow v}{f(e_1, \ldots, e_n) \Downarrow v}$$

We will abbreviate $e_1 \Downarrow v_1, \ldots, e_n \Downarrow v_n$ with $\mathbf{e} \Downarrow \mathbf{v}$.

3.2 Reduction semantics

The (internal) behaviour of a program is specified by (i) a *reduction system* $\rightarrow$ describing the possible reductions of the program *during an instant* and (ii) an *evaluation relation* $\mapsto$ determining how a program evolves at the *end of each instant*. These definitions rely on a structural equivalence relation $\equiv$ that we introduce first.

3.2.1 Structural equivalence

The *structural equivalence* $\equiv$ is the least equivalence relation on programs that identifies programs up to α-renaming and that satisfies the following standard equations:

$$P \mid 0 \equiv P, \qquad P_1 \mid P_2 \equiv P_2 \mid P_1, \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3),$$
$$\nu s\ P \equiv P \text{ if } s \notin fn(P), \qquad \nu s\ P_1 \mid P_2 \equiv \nu s\ (P_1 \mid P_2) \text{ if } s \notin fn(P_2)\ .$$

3.2.2 Reduction relation

We introduce the following *reduction rules*:

$$\frac{e \Downarrow v}{\overline{s}e \mid s(x).P, K \rightarrow\!\triangleright\ \overline{s}e \mid [v/x]P} \qquad \frac{A(\mathbf{x}) = P \quad \mathbf{e} \Downarrow \mathbf{v}}{A(\mathbf{e}) \rightarrow\!\triangleright\ [\mathbf{v}/\mathbf{x}]P}$$

$$\frac{}{[s = s]P_1, P_2 \rightarrow\!\triangleright\ P_1} \qquad \frac{s \neq s'}{[s = s']P_1, P_2 \rightarrow\!\triangleright\ P_2}$$

$$\frac{match(v, p) = \sigma}{[v \trianglerighteq p]\ P_1, P_2 \rightarrow\!\triangleright\ \sigma P_1} \qquad \frac{match(v, p) \text{ undefined}}{[v \trianglerighteq p]\ P_1, P_2 \rightarrow\!\triangleright\ P_2}$$

A *static context* C is defined by $C ::= [\ ] \mid \nu s\ C \mid (C \mid P)$. The *reduction relation* $\rightarrow$ is then defined by the rule:

$$\frac{P \equiv C[P'] \quad P' \rightarrow\!\triangleright\ Q' \quad C[Q'] \equiv Q}{P \rightarrow Q}$$

3.2.3 Suspension and evaluation at the end of the instant

We write $P \downarrow$ if $\neg\exists Q\ (P \to Q)$ and say that the program P is *suspended*. When P is suspended the instant ends and an additional computation is carried on to move to the next instant. This goes in three steps that amount to: (1) collect in lists the set of values emitted on every signal, (2) extrude the signal names contained in values visible at the end of the instant, and (3) initialise the continuations K of the present statements.

To this end, we introduce first some notation. A suspended program P is structurally equivalent to:

$$\nu\mathbf{s}(S \mid In) \tag{1}$$

where the signal names $\mathbf{s}$ are all distinct, $S \equiv \overline{s_1}e_1 \mid \cdots \mid \overline{s_n}e_n$, $In \equiv t_1(x_1).P_1, A_1(\mathbf{r_1}) \mid \cdots \mid t_m(x_m).P_m, A_m(\mathbf{r_m})$, and $n, m \geq 0$ (by convention an empty parallel composition equals the program 0). We write $\overline{s}e \in S$ to mean that $\overline{s}e$ occurs in the parallel composition S. We can now formalise the steps (1–3).

1. Let V be a function from signal names to lists of values. We say that V *represents* S and write $V \Vdash S$ if for all signal names s, if $\{v_1, \ldots, v_n\} = \{v \mid \overline{s}e \in S,\ e \Downarrow v\}$ then $V(s) = [v_{\pi(1)}; \cdots ; v_{\pi(n)}]$ for some permutation π.

2. We define $\mathit{Free}(\nu\mathbf{s}\ S)$ as the least set of signal names such that $\mathit{Free}(\nu\mathbf{s}\ S) \supseteq \mathit{fn}(\nu\mathbf{s}\ S)$ and if $s \in \mathit{Free}(\nu\mathbf{s}\ S)$, $\overline{s}e \in S$, $e \Downarrow v$, and $s' \in \mathit{fn}(v)$ then $s' \in \mathit{Free}(\nu\mathbf{s}\ S)$. For instance, $\mathit{Free}(\nu s_1, s_2\ \overline{s}s_1 \mid \overline{s_1}s_2) = \{s, s_1, s_2\}$.

3. If r is an expression with dereferenciation then $V(r)$ is the expression resulting from the replacement of all dereferenced signals $!s$ with $V(s)$. If $A(\mathbf{r})$ is a continuation K of a present statement, where $\mathbf{r}$ are closed expressions, then $\mathit{Eval}(A(\mathbf{r}), V) = A(\mathbf{v})$ if $V(\mathbf{r}) \Downarrow \mathbf{v}$. Finally, if In is defined as in (1) then $\mathit{Eval}(In, V) = \mathit{Eval}(A_1(\mathbf{r}_1), V) \mid \cdots \mid \mathit{Eval}(A_m(\mathbf{r}_m), V)$.

With these conventions, we can now state the evaluation rule at the end of the instant:

$$\frac{P \downarrow \quad P \equiv \nu\mathbf{s}\ (S \mid In) \quad V \Vdash S \quad \{\mathbf{s}'\} = \{\mathbf{s}\} \backslash \mathit{Free}(\nu\mathbf{s}\ S) \quad P' \equiv \nu\mathbf{s}'\mathit{Eval}(In, V)}{P \mapsto P'}$$

In this rule, (i) we decompose the suspended program in emissions and inputs, (ii) we compute a representation of the emission, (iii) we compute the signal names extruded, and finally (iv) we remove the emitted names and initialise the continuations of the present statements.

3.3 Feasible reactivity

At the beginning of each instant, a program receives an input that we may represent as a (fresh) thread identifier Env defined by the equation $\mathit{Env}() = \overline{s_1}v_1 \mid \cdots \mid \overline{s_n}v_n$. Then we write

$$P \overset{\mathit{Env}}{\mapsto} P' \text{ if } P \mapsto P'' \text{ and } P' \equiv (P'' \mid \mathit{Env})$$

By the properties of the model, we may assume without loss of generality that in the input all values emitted on a signal s are *distinct*.

Definition 1 (computation). *A computation of a program P is an infinite and countable sequence of programs $P_1, P_2, \ldots$ such that*

$$P \equiv P_1 \xrightarrow{*} P_{i_1} \overset{Env_1}{\mapsto} P_{i_1+1} \xrightarrow{*} P_{i_2} \overset{Env_2}{\mapsto} P_{i_2+1} \cdots$$

In general, the reduction of $P_1, P_{i_1+1}, P_{i_2+1}, \ldots$ may fail to reach the end of the instant. We call *reactive* the programs that are guaranteed to suspend.

Definition 2 (reactivity). *A program P is reactive if in all computations that start with P, the evaluation at the end of the instant occurs infinitely often.*

Example 4. *With reference to the example 3, a possible computation of the program $A(s) \mid C(s)$ is as follows:*

$$\begin{array}{lll} A(s) \mid C(s) & \xrightarrow{*} \mathsf{pause}.B(s, !s) \mid \nu n_0\ \overline{s} n_0 \mid \mathsf{pause}.C(s) & \overset{Env_1}{\mapsto} B(s, [n_0]) \mid C(s) \\ & \xrightarrow{*} \overline{s} n_0 \mid \mathsf{pause}.B(s, !s) \mid \nu n_1\ \overline{s} n_1 \mid \mathsf{pause}.C(s) & \overset{Env_2}{\mapsto} B(s, [n_0; n_1]) \mid C(s) \cdots \end{array}$$

In this case, we assume that the input at the beginning of each instant is empty, $Env_i() = 0$ for $i = 1, 2, \ldots$. Note that the order of the signal names in the list ℓ, which is a parameter of the identifier B, is chosen non-deterministically at the beginning of each instant.

We assume that initially a program has the shape

$$\nu \mathbf{s}(A_1(\mathbf{v_1}) \mid \cdots \mid A_n(\mathbf{v_n})) \tag{2}$$

Then, by the definition of the present instruction and the input, a program will have this shape at the beginning of each instant, up to structural equivalence. The definition of feasible reactivity is relative to the size of the initial program and the size of the (largest) input. By convention, the size of a program with the shape (2) is n plus the sum of the sizes of the values $\mathbf{v_1}, \ldots, \mathbf{v_n}$. The size of an input *Env* defined by an equation $Env = \overline{s_1} v_1 \mid \cdots \mid \overline{s_n} v_n$ is the size of the list $[v_1; \ldots; v_n]$.

Definition 3 (feasible reactivity). *A program P of the shape (2) is* feasibly reactive *if there exists a polynomial Q such that for every computation*

$$P \equiv P_{i_0+1} \xrightarrow{*} P_{i_1} \overset{Env_1}{\mapsto} P_{i_1+1} \xrightarrow{*} P_{i_2} \overset{Env_2}{\mapsto} P_{i_2+1} \cdots$$

if d bounds the size of P and the sizes of $Env_1, \ldots, Env_k$ for $k \geq 1$ then (i) P_{i_k+1} (the program at the beginning of the instant k) has size bounded by $Q(d)$ and (ii) it is guaranteed to suspend in time less than $Q(d)$,

For instance, the program in example 3 fails to be feasibly reactive because the size of the parameter ℓ of the identifier B grows by one every instant.

4 Annotations and Constraints Generation

Programs come with a finite system of recursive equations. Our static analysis actually concerns this system and it is independent of the particular program that is used to initialise the computation. The reader should keep in mind that the analysis of a program is actually the analysis of the associated system. We restrict our attention to *finite control programs*. To this end, we inspect the system of equations and we check that in each equation $A(\mathbf{x}) = P$, P cannot spawn two recursive calls that run in parallel. Also, the static analysis makes abstraction of the actual signal names while keeping track of the region they belong to. It will be convenient to suppose that the program does not contain trivial matchings such as a value matching a pattern ($[v \trianglerighteq p]\ P_1, P_2$) and the comparison of two identical names ($[s = s]\ P_1, P_2$). Such matchings can be removed by a trivial symbolic execution.

4.1 Reset annotations and read once condition

We denote with *Reset* a subset of the thread identifiers containing those thread identifiers that correspond to the beginning of a new 'cycle'. To be in *Reset* a thread identifier A has to satisfy one of the following conditions: either it is defined by an equation of the shape $A(\ldots) = \mathsf{pause}.K$ or all its occurrences in the program are in the else branch of a present statement. By these syntactic conditions, we guarantee that the end of a cycle for a given thread always entails the end of its computation for the current instant. For instance, in the example 2, it is natural to assume that $\mathit{Server} \in \mathit{Reset}$ and $\mathit{Handle} \notin \mathit{Reset}$.

As we have seen, a program may *read* a signal during an instant with the present statement or at the end of the instant through dereferencing. The *read once condition* is the hypothesis that for every thread, in every cycle, there is a bound on the number of times the reading of a signal can be performed. Specifically, we require and statically check on the *call graph* of the program (see below) that the computation performed starting from any thread identifier can execute any given read instruction at most once within a cycle.

1. We assign to every present statement and to every dereferencing in a program a distinct fresh label (a variable), y, and we collect all these labels in an ordered sequence, $y_1, \ldots, y_m$. In the following, we will use the notation $s^y(x).P, K$ and $!^y s$ to make the labels explicit. If $\mathbf{r}$ is a vector of expressions with dereferenciation, we denote with $\mathit{Lab}(\mathbf{r})$ the finite set of labels that occur in $\mathbf{r}$.

2. With every thread identifier A defined by an equation $A(\mathbf{x}) = P$, we associate a node of the graph. We also introduce a fresh thread identifier O and a related node that plays the role of a *sink* in the call graph.

3. We define a function *Call* that takes in input a program and a finite set of labels and produces in output a finite set of pairs composed of a thread identifier and a set of

labels. The function *Call* is defined as follows:

$$
\begin{array}{lcl}
Call(0, L) & = & \{(O, L)\} \\
Call(\overline{s}e, L) & = & \{(O, L)\} \\
Call(s^y(x).P, A(\mathbf{r}), L) & = & \begin{cases} Call(P, L \cup \{y\}) \cup \{(A, L \cup Lab(\mathbf{r}))\} & \text{if } A \notin Reset \\ Call(P, L \cup \{y\}) \cup \{(O, L \cup Lab(\mathbf{r}))\} & \text{otherwise} \end{cases} \\
Call(A(\mathbf{e}), L) & = & \begin{cases} \{(A, L)\} & \text{if } A \notin Reset \\ \{(O, L)\} & \text{otherwise} \end{cases} \\
Call([s_1 = s_2]\ P_1, P_2, L) & = & Call(P_1, L) \cup Call(P_2, L) \\
Call([x \trianglerighteq p]\ P_1, P_2, L) & = & Call(P_1, L) \cup Call(P_2, L) \\
Call(P_1 \mid P_2, L) & = & Call(P_1, L) \cup Call(P_2, L) \\
Call(\nu s\ P, L) & = & Call(P, L)
\end{array}
$$

4. Suppose the identifier A is defined by an equation $A(\mathbf{x}) = P$ and that $C = Call(P, \emptyset)$. We introduce an edge from A to an identifier B (possibly O) if $(B, L) \in C$. In this case, we label the edge with the set $\bigcup\{L \mid (B, L) \in C\}$.

5. We denote with $R(A)$ the union of the sets of labels of the edges accessible from A and with $\mathbf{y}_A$ the ordered sequence of labels in $R(A)$.

The definition of *Call* is such that for every sequence of calls in the execution of a thread within the cycle we can find a corresponding path in the call graph.

Definition 4 (read once condition). *A program satisfies the read once condition if in the call graph there are no loops that go through an edge whose label is a non-empty set.*

Note that while the number of reads is bounded by a constant, the amount of information that can be read is not. Thus, for instance, a 'server' thread can just read one signal in which is stored the list of requests produced so far and then it can go on scanning the list and replying to all the requests within the same instant. In the following, we will focus on programs that satisfy the read once condition. For such programs, we introduce for each thread identifier A with parameters $\mathbf{x}$, a fresh thread identifier A^+ whose parameters are those of A plus the parameters $\mathbf{y}_A$ that can be read within a cycle. The idea is that the behaviour generated by the thread identifier A within a cycle can be described as a *function* of its parameters $\mathbf{x}$ which are determined at the beginning of the cycle and the values $\mathbf{y}_A$ of the signals read within the cycle. We will also refer to $\mathbf{x}$ as *proper parameters* and to $\mathbf{y}_A$ as *auxiliary parameters* of the identifier A^+.

Example 5. *Consider example 1 and suppose that Cell $\in$ Reset marks the end of a cycle and that the label associated with the dereferenciation is y. The graph resulting from the analysis has three nodes $\{Cell, Send, O\}$ and the following labelled edges: $(Cell, \emptyset, Send)$, $(Send, \emptyset, Send)$ and $(Send, \{y\}, O)$. The program satisfies the read once condition since the only possible loop, namely the one form Send to Send, is composed of edges (just one in this case) whose label is the empty set. Both $Send^+$ and $Cell^+$ have an auxiliary parameter y.*

Next consider example 2 and suppose that Server $\in$ Reset and the label associated with the dereferenciation is y. The call graph has three nodes Server, Handle, O and the following labelled edges: $(Server, \{y\}, Handle)$, $(Handle, \emptyset, Handle)$, and $(Handle, \emptyset, O)$. Again the read once condition is satisfied. $Server^+$ has an additional parameter y while $Handle^+$ has no additional parameter.

Finally, consider example 3 and suppose that $A, C \in$ Reset and the label associated with the dereferenciation is y. The graph has four nodes: C, A, B, O and the following labelled edges: $(A, \{y\}, B)$, $(B, \emptyset, B)$, $(B, \emptyset, O)$, and $(C, \emptyset, C)$. In this case too, the read once condition is satisfied. A^+ has an additional parameter y while B^+ and C^+ have no additional parameter.

4.2 Status annotations

We associate a *status*, either *lexicographic* (*lex*) or *multi-set* (*mset*), with every thread identifier. We assume that thread identifiers which are equivalent with respect to a pre-order $\geq_F$ that we define below have the same arity and the same status. We note that this implies that A^+, B^+ have the same arity too.

To define the pre-order $\geq_F$, we introduce first a *call graph within an instant* by modifying the definition given in section 4.1 so that $Call(A(\mathbf{e}), L) = \{(A, L)\}$ and $Call(s(x).P, K, L) = Call(P, L)$. Thus there is an edge from the identifier A to the identifier B if in the definition of A, say $A(\mathbf{x}) = P$, it is possible to call B within the same instant A is called. Second, we build the least pre-order (reflexive and transitive) $\geq_F$ over thread identifiers such that $A \geq_F B$ if there is an edge from A to B in the call graph within an instant. We write $A =_F B$ if $A \geq_F B$ and $B \geq_F A$, and $A >_F B$ if $A \geq_F B$ and $A \neq_F B$. The rank of the thread identifier A, noted $rank(A)$, is the length of the longest chain $A >_F B >_F \ldots$

4.3 Parameter annotations

One of our goals is to control the size of the proper parameters of a thread. However, it is sometimes appropriate to *neglect* some parameters. For instance, consider the example 2. One of the parameters of the thread identifier *Handle* is a list ℓ that is read on a signal s whose size is unrelated to the size of the parameter s of the thread identifier *Server*. We observe that the parameter ℓ is needed by *Handle* to perform some computation and that this parameter is then neglected at the end of the cycle. We then introduce a mechanism to *mask* parameters such as ℓ. Let `0` be a fresh constant that stands for a parameter of size 0. If h is a function of arity n and $I \subseteq \{1, \ldots, n\}$ is a subset of its parameters then $h(e_1, \ldots, e_n)_I$ is defined as $h(e'_1, \ldots, e'_n)$ where $e'_i = e_i$ if $i \in I$ and $e'_i =$ `0` otherwise. Intuitively, in $h(e_1, \ldots, e_n)_I$ 'we set to 0' all arguments that are not in I. For each thread identifier A defining a behaviour of arity n, we assume a set $I_A \subseteq \{1, \ldots, n\}$ with the condition that $I_A = \{1, \ldots, n\}$ if A marks the end of a cycle in the program (thus in the latter case, no parameter can be set to 0). Note that the mask acts only on the proper parameters of the identifier A and *not* on the auxiliary parameters $\mathbf{y}_A$ corresponding to the values read within a cycle.

4.4 Signal annotations

One purpose of the signal annotations is to reject programs such as the one in example 3. Let us consider in particular the thread A. At each instant, this thread re-emits on a signal s the values emitted on the *same* signal s at the previous instant. We want to reject this kind of behaviour while allowing –under suitable conditions– a slightly different behaviour where a thread emits on a signal s a series of values (possibly the same) that depend on the values emitted on a *different* signal s' at the previous instant. For instance, we want to be able to program a 'server' (cf. example 2) that receives a series of requests at the end of the instant and produces a series of related answers in the following instant. The idea is to partition the signal names into a finite collection of *regions*. Then regions are ordered and the behaviour of the server described above is allowed if the signal s belongs to a region that is strictly below the region to which s' belongs. For instance, in the example 2, we emit on signal s' a value which depends on a value read on a signal s. If we admit that this value has arbitrary size then we should require that the signal s is associated with a region smaller than the region associated with s'.

Formally, we assume a set of regions $\mathcal{R} = \{\rho_1, \rho_2, \ldots\}$ with a strict order $>_{\mathcal{R}}$ and we denote with $rank(\rho)$ the length of the longest sequence $\rho >_{\mathcal{R}} \rho_1 >_{\mathcal{R}} \cdots >_{\mathcal{R}} \rho_n$. We assume that every signal type comes with a region annotation $Sig_\rho(t)$ so that the type of a signal name also provides the region to which the signal name belongs. In section 4.6, we will rely on these annotations to derive inequalities that guarantee that the size of the values emitted on a signal of region ρ can be bound as a function of the size of the values received on signals belonging to regions of smaller rank.

4.5 Inequalities

We rely on the annotations to produce a set of inequalities. We use the notation $\overline{\mathbf{r}}$ for $\mathbf{r}$ where each $!^y s$ is replaced with y. Given a system of equations, for each thread identifier A defined by an equation $A(\mathbf{x}) = P$, we compute $\mathcal{C}_i(P, A^+(\mathbf{x}, \mathbf{y}_A))$, with index $i = 0, 1, 2$ according to the rules described in table 1. The definition of the functions $\mathcal{C}_i$ amounts to perform a 'symbolic execution' of the body P of the equation while keeping track of the shape of the parameters $\mathbf{x}$ and the values read $\mathbf{y}_A$. More precisely, the functions $\mathcal{C}_i$ explore the finitely many control points of a computation starting with a recursive call to the thread identifier A. At some critical points, namely (i) when a value is emitted, (ii) when a value is received, and (iii) when a recursive call is executed, the functions $\mathcal{C}_i$ produce certain inequalities whose purpose is discussed next.

4.5.1 Inequalities for termination of the instants

In our model, the only way a computation may fail to be reactive is that a thread goes through a recursive call infinitely often within an instant. To avoid this situation, we have to make sure that whenever the identifiers $A_1, \ldots, A_n$ may call each other, a certain well-founded measure decreases. This is the purpose of the inequalities of index 0. Moreover,

$$
\begin{array}{ll}
\mathcal{C}_i(P, A^+(\mathbf{p})) & = \mathsf{case}\ P\ \mathsf{of} \\
0 & : \emptyset \\
[x \trianglerighteq p]\ P_1, P_2 & : \mathcal{C}_i(P_1, A^+([p/x]\mathbf{p})) \cup \mathcal{C}_i(P_2, A^+(\mathbf{p})) \\
[s_1 = s_2]P_1, P_2 & : \mathcal{C}_i(P_1, A^+(\mathbf{p})) \cup \mathcal{C}_i(P_2, A^+(\mathbf{p})) \\
(P_1 \mid P_2) & : \mathcal{C}_i(P_1, A^+(\mathbf{p})) \cup \mathcal{C}_i(P_2, A^+(\mathbf{p})) \\
\nu s\ P' & : \mathcal{C}_i(P', A^+(\mathbf{p})) \\
\overline{s}e, i = 0, 1 & : \emptyset \\
\_, i = 2 & \{A^+(\mathbf{p})_{\downarrow\rho} \geq_2 e\} \qquad s : Sig_\rho(t) \\
B(\mathbf{e}), i = 0 & : \begin{cases} \emptyset & \text{if } A >_F B \\ \{A^+(\mathbf{p}) >_0 B^+(\mathbf{e}, \mathbf{y}_B)\} & \text{otherwise} \end{cases} \\
\_, i = 1 & : \{A^+(\mathbf{p})_{I_A} \geq_1 B^+(\mathbf{e}, \mathbf{y}_B)_{I_B}\} \\
\_, i = 2 & : \begin{cases} \{A^+(\mathbf{p})_{\downarrow\rho} \geq_2 B^+(\mathbf{e}, \mathbf{y}_B)_{\downarrow\rho} \mid \rho \in \mathcal{W}(B)\} & B \notin \mathit{Reset} \\ \emptyset & \text{otherwise} \end{cases} \\
s^y(x).P', B(\mathbf{r}), i = 0 & : \mathcal{C}_0([y/x]P', A^+(\mathbf{p})) \\
\_, i = 1 & : \mathcal{C}_1([y/x]P', A^+(\mathbf{p})) \cup \{A^+(\mathbf{p})_{I_A} \geq_1 B^+(\overline{\mathbf{r}}, \mathbf{y}_B)_{I_B}\} \\
\_, i = 2 & : \mathcal{C}_2([y/x]P', A^+(\mathbf{p})) \\
 & \cup \begin{cases} \{A^+(\mathbf{p})_{\downarrow\rho} \geq_2 B^+(\overline{\mathbf{r}}, \mathbf{y}_B)_{\downarrow\rho} \mid \rho \in \mathcal{W}(B)\} & B \notin \mathit{Reset} \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

Table 1: Inequalities of index $0, 1, 2$

the inequalities will be interpreted so as to make sure that a decrement step can only be taken polynomially many times in the size of the values.

Example 6. *We rely on the call graphs computed in example 5. For the example 1, we obtain:* $Send^+(s,q,\ell,\mathsf{cons}(s',\ell''),y) >_0 Send^+(s,q,\ell,\ell'',y)$*, for the example 2, we obtain:* $Handle^+(s,\mathsf{cons}(\mathsf{req}(s',x),\ell')) >_0 Handle^+(s,\ell')$*, and for the example 3, we obtain:* $B^+(s,\mathsf{cons}(n,\ell')) >_0 B^+(s,\ell')$.

4.5.2 Inequalities for size control at the beginning of a cycle

The purpose of the inequalities of index 1 is to ensure that the size of the parameters of a thread at the *beginning of a new cycle* is bounded by a function (a polynomial) of the size of the initial parameters of the computation. Of course, a cycle starting with A may span several instants and may go through several recursive calls before a new cycle is started again. For this reason, the *invariant* we have to maintain concerns *all* recursive calls both *within* and *at the end* of the instant.

Example 7. *We rely again on the computation of the call graphs in example 5. For example 1, assuming* $I_{Cell} = \{1,2,3\}$ *and* $I_{Send} = \{1,2,3,4\}$ *we obtain:*

$$Cell^+(s,q,\ell,0) \geq_1 Send^+(s,q,\ell,\ell,0), \quad Send^+(s,q,\ell,\ell',0) \geq_1 Cell^+(s,next(q,y),\ell,0),$$
$$Send^+(s,q,\ell,\mathsf{cons}(s',\ell''),0) \geq_1 Send^+(s,q,\ell,\ell'',0)\ .$$

For example 2, assuming $I_{Server} = I_{Handle} = \{1\}$ *we obtain:*

$$Server^+(s,0) \geq_1 Handle^+(s,0), \qquad Handle^+(s,0) \geq_1 Handle^+(s,0),$$
$$Handle^+(s,0) \geq_1 Server^+(s,0)\ .$$

For example 3, assuming $I_A = I_B = I_C = \{1\}$*, we obtain:*

$$A^+(s,0) \geq_1 B^+(s,0), \quad B^+(s,0) \geq_1 B^+(s,0), \quad B^+(s,0) \geq_1 A^+(s,0), \quad C^+(s) \geq_1 C^+(s)\ .$$

4.5.3 Inequalities for size control within a cycle

Finally, the purpose of the inequalities of index 2 is to ensure that the size of any value emitted during a cycle in a given region as well as the number of these emissions within an instant is polynomial in the size of the parameters at the beginning of the cycle, the inputs provided by the environment, and the size of the values read in regions of smaller rank.

1. Given a thread identifier A, we compute an over approximation of the set of regions associated with an output within a cycle starting from A. To this end, we use the call graph defined in section 4.1 and we compute all thread identifiers that are reachable from A within a cycle. Then we inspect the definition of each thread identifier (different from O) and determine the regions associated with the emissions that may arise in the definition. We denote with $\mathcal{W}'(A)$ this set. Moreover, let $\rho_\top$ be a region whose rank is higher than the rank of all the regions used in the program and let $\mathcal{W}(A)$ equal $\{\rho_\top\}$ if $\mathcal{W}'(A) = \emptyset$, and $\mathcal{W}'(A)$ otherwise.

2. Let A be a thread identifier of arity n with auxiliary parameters $\mathbf{y}_A = y_1, \ldots, y_m$. We can associate with each position $1, \ldots, m$ in the list of auxiliary parameters a unique region $\gamma(i)$ which is the region associated with the corresponding read instruction. Given a region ρ, we denote with $\downarrow \rho$ the set of regions of rank smaller than ρ. In particular, if $\mathit{rank}(\rho) = 0$ then we $\downarrow \rho = \emptyset$. Given a set M of regions we introduce the notation $A^+(\mathbf{p})_M$ for $A^+(\mathbf{p})_I$ where $I = \{1, \ldots, n\} \cup \{n + i \mid \gamma(i) \in M\}$. Thus, this amounts to set to 0 all auxiliary parameters whose region is not in M. Note that this masking only affects the *auxiliary* parameters of the thread identifiers.

Example 8. *Consider example 1, assuming all the signals on which the automata interact belong to the same region ρ. In this case, $\mathcal{W}(Cell) = \mathcal{W}(Send) = \{\rho\}$ and the resulting inequalities are:*

$$\begin{gathered} Cell^+(s, q, \ell, 0) \geq_2 Send^+(s, q, \ell, \ell, 0), \\ Send^+(s, q, \ell, \mathsf{cons}(s', \ell''), 0) \geq_2 Send^+(s, q, \ell, \ell'', 0), \quad Send^+(s, q, \ell, \mathsf{cons}(s', \ell''), 0) \geq_2 q \ . \end{gathered}$$

Next consider example 2, assuming the region ρ of the signal on which the Server receives the requests is below the region ρ' of the signals on which it provides an answer. In this case, $\mathcal{W}(Server) = \mathcal{W}(Handle) = \{\rho'\}$ and the resulting inequalities are:

$$\begin{gathered} Server^+(s, y) \geq_2 Handle^+(s, y), \quad Handle^+(s, \mathsf{cons}(\mathsf{req}(s', x), \ell')) \geq_2 Handle^+(s, \ell'), \\ Handle^+(s, \mathsf{cons}(\mathsf{req}(s', x), \ell')) \geq_2 f(x) \ . \end{gathered}$$

Finally, consider example 3. Here we have just one signal belonging, say, to a region ρ. In this case, $\mathcal{W}(A) = \mathcal{W}(B) = \mathcal{W}(C) = \{\rho\}$ and the resulting inequalities are:

$$\begin{gathered} A^+(s, 0) \geq_2 B^+(s, y), \quad B^+(s, \mathsf{cons}(n, \ell')) \geq_2 B^+(s, \ell'), \quad B^+(s, \mathsf{cons}(n, \ell')) \geq_2 n, \\ C^+(s) \geq_2 C^+(s), \quad C^+(s) \geq_2 n \ . \end{gathered}$$

We anticipate that the inequality $A^+(s, 0) \geq_2 B^+(s, y)$ is not going to be satisfiable since A does not depend on y which is a list of arbitrary size.

4.6 Assignments and quasi-interpretations

We introduce first the notion of *assignment* which interprets the inequalities in terms of certain numerical functions. A *quasi-interpretation* is then an assignment that *satisfies* the inequalities associated with the program.

4.6.1 Assignments

Let h denote either a constructor c or a function symbol f or a thread identifier A^+. An *assignment* associates with each symbol h of arity n of the program a function $q_h : \mathbb{N}^n \to \mathbb{N}$ subject to a series of conditions that we specify below.

First we have to introduce some notation. Let E denote a formula which is either an expression e or the application of a thread identifier to expressions $A^+(e_1, \ldots, e_n)$. Suppose

E contains the variables $x_1, \ldots, x_n$. Once an assignment is fixed, we can associate with E a function over the natural numbers of arity n by defining $q_{x_i} = x_i$ and $q_{h(e_1,\ldots,e_n)} = q_h(q_{e_1}, \ldots, q_{e_n})$. In particular, we note that if v is a value then q_v is a numerical constant.

A *ground substitution* is a substitution that associates values with variables (while respecting the types). Given two formulae E_1, E_2, we write $q \models E_1 > E_2$ ($q \models E_1 \geq E_2$) if for all ground substitutions σ, $q_{\sigma E_1} > q_{\sigma E_2}$ ($q_{\sigma E_1} \geq q_{\sigma E_2}$).[1]

We will also compare *vectors* of formal expressions. For *lexicographic comparison*, we write $q \models (E_1, \ldots, E_n) >_{lex} (E'_1, \ldots, E'_n)$ if there is an $i \leq n$ such that $q \models E_j \geq E'_j$ for $j = 1, \ldots, i-1$ and $q \models E_i > E'_i$. For *multi-set comparison*, we write $q \models (E_1, \ldots, E_n) >_{mul} (E'_1, \ldots, E'_n)$ if for all ground substitutions σ, $\{\!|q_{\sigma E_1}, \ldots, q_{\sigma E_n}|\!\} >^{\mathbb{N}}_{mset} \{\!|q_{\sigma E'_1}, \ldots, q_{\sigma E'_n}|\!\}$, where $\{\!| \ldots |\!\}$ is our notation for multi-sets and $>^{\mathbb{N}}_{mset}$ is the well-founded multi-set order over the finite multi-sets of natural numbers. We notice the following simple combinatorial fact about lexicographic and multi-set orders which is instrumental to establish *polynomial time* termination.

Lemma 1. *Suppose $a_1, \ldots, a_n, c$ are natural numbers and $a_1, \ldots, a_n < c$. Then the length of any strictly decreasing sequence of the shape $(a_1, \ldots, a_n) >_{lex} (b_1, \ldots, b_n) >_{lex} \cdots$ or of the shape $\{\!|a_1, \ldots, a_n|\!\} >^{\mathbb{N}}_{mset} \{\!|b_1, \ldots, b_n|\!\} >^{\mathbb{N}}_{mset} \cdots$ is bounded by c^n.*

Definition 5. *An assignment should satisfy the following conditions.*

(1) *If s is a signal name or c is a constructor with arity 0 then $q_s = q_{\mathsf{c}} = 0$. Otherwise, if c is a constructor with positive arity n then $q_{\mathsf{c}} = d_{\mathsf{c}} + \Sigma_{i=1,\ldots,n} x_i$ for some natural number $d_{\mathsf{c}} \geq 1$.*

(2) *For all symbols h of arity n it holds that: (i) $q \models h(x_1, \ldots, x_n) \geq x_i$ for $i = 1, \ldots, n$ and (ii) q_h is monotonic,* i.e., *$a_j \geq b_j$ for $j = 1, \ldots, n$ implies $q_h(a_1, \ldots, a_n) \geq q_h(b_1, \ldots, b_n)$.*

(3) *Let f be a function symbol of arity n. Then $f(v_1, \ldots, v_n) \Downarrow v$ implies that $q_{f(v_1,\ldots,v_n)} \geq q_v$.*

It follows from condition (1) that there is a constant $k \geq 1$ (that can be taken as the largest additive constant d_{c}) such that for any value v, $|v| \leq q_v \leq k \cdot |v|$. Also note that condition (1) implies condition (2) for constructors.

The definition of an assignment q ensures that $\sigma e \Downarrow v$ implies $q_{\sigma e} \geq q_v$. We say that a function $U : \mathbb{N} \to \mathbb{N}$ bounds the assignment q if for all symbols h and all natural numbers n it holds that $q_h(n, \ldots, n) \leq U(n)$. We say that an assignment is polynomially bounded if it can be bound by a function U which is a polynomial. In the following, we will restrict our attention to polynomially bounded assignments.

[1] Sometimes, a stronger definition of satisfaction is considered that requires *e.g.*, $q_{E_1} \geq q_{E_2}$ where q_{E_1}, q_{E_2} are regarded as functions over the natural numbers. We prefer the definition based on ground substitutions because it allows to exploit some information on the data size. For instance, we may satisfy a constraint $f(x) \geq \mathsf{c}(x, y)$ if we know that all the values that may replace y have bounded size. On the other hand, with the stronger definition such constraint cannot be satisfied.

4.6.2 Quasi-interpretations

A quasi-interpretation is a polynomially bounded assignment which satisfies the constraints of index $0, 1, 2$.

Definition 6 (quasi-interpretation). *An assignment q is a quasi-interpretations if:*
(1) *For all constraints of the shape $A^+(p_1, \ldots, p_n) >_0 B^+(e_1, \ldots, e_n)$ where $A =_F B$, with status st, we have:*

$$q \models (p_1, \ldots, p_n) >_{st} (e_1, \ldots, e_n) \ .$$

(2) *For all constraints of the shape $A^+(p_1, \ldots, p_n) \geq_i B^+(e_1, \ldots, e_m)$ $(i = 1, 2)$ and $A^+(p_1, \ldots, p_n) \geq_2 e$ we have:*

$$q \models A^+(p_1, \ldots, p_n) \geq B^+(e_1, \ldots, e_m) \text{ and } q \models A^+(p_1, \ldots, p_n) \geq e \ .$$

Example 9. *Consider example 1 and assume that we attribute the lexicographic status to Cell and Send. We note that Cell $>_F$ Send. The inequality of index 0 is satisfied because the quasi-interpretation of $\mathsf{cons}(s', \ell'')$ is always strictly larger than the quasi-interpretation of ℓ''. To satisfy the remaining inequalities of index $1, 2$ it suffices to interpret $Cell^+$ and $Send^+$ as the maximum function, noticing that $next(q, y)$ is always a state which is represented by a constant of size 0.*

Next, consider example 2 and assume lexicographic status for the thread identifiers. We note that Handle $>_F$ Server. Again the inequality of index 0 is satisfied because the quasi-interpretation of $\mathsf{cons}(\mathsf{req}(s', x), \ell')$ is always larger than the quasi-interpretation of ℓ'. To satisfy the inequalities of index $1, 2, 3$ it suffices to suppose that the quasi-interpretation of $Handle^+$ and $Server^+$ is a function $g : \mathbb{N}^2 \to \mathbb{N}$ such that $g(0, x)$ is pointwise larger than the quasi-interpretation of the function f. Finally, consider example 3. We note that $A >_F B$. We can satisfy the inequalities of index $0, 1$ but as anticipated there is no way the inequality $A^+(s, 0) \geq_2 B^+(s, y)$ can be satisfied since y ranges over lists of arbitrary size.

We can now state our main result whose proof will be discussed in the following section 5.

Theorem 1. *A program that admits a polynomial quasi-interpretation is feasibly reactive.*

5 Proofs outline

We are given a finite system of recursive equations. The initial configuration of a program relatively to such a system has the shape: $R = \nu \mathbf{s}\ (A_1(\mathbf{v}_1) \mid \cdots \mid A_n(\mathbf{v}_n))$. Since we have assumed that the system is *finite control* during the computation we will have at most n main parallel threads plus a variable number of auxiliary threads that may just branch and emit signals and that disappear at the end of each instant. Of course one of our goals is to show that this variable number of threads can be polynomially bounded.

Lemma 2. *Let R be a program admitting a polynomial quasi-interpretation. There is a polynomial $Q(x)$ such that if c bounds the size of R, the size of the inputs, and the sizes of the parameters of all calls within a given instant then the program in that instant will suspend in time less than $Q(c)$.*

The computation performed by the program is simply the interleaving of the computations performed by the n main threads. It is clear that the computation a thread may perform within an instant before running a recursive call is polynomially bounded in c. Thus it is enough to show that each thread may perform at most polynomially many recursive calls before suspending and to this end we rely on the inequalities of index 0 and the lemma 1. Note that the size and the number of the values emitted during the instant is polynomial in c and that therefore their concatenation in a list has size polynomial in c too. We anticipate that the proof we have sketched of the lemma 2 actually shows that each thread whose parameters and inputs are bound by c will suspend in time polynomial in c.

Lemma 3. *Let R be a program admitting a polynomial quasi-interpretation. There is a polynomial $Q(x)$ such that if c bounds the size of R and $A \in Reset$ then, in all computations of R, the sizes of the parameters in every call to A are bounded by $Q(c)$.*

The inequalities of index 1 guarantee that a computation that starts with $B(\mathbf{v})$ will have the property that $B(\mathbf{v})$ will 'dominate' (up to quasi-interpretation and modulo the parameter annotations) all the following calls $A(\mathbf{u})$ including those that correspond to a reset point and in this case all parameters of the call are taken into account by the definition of I_A.

Lemma 4. *Let R be a program admitting a polynomial quasi-interpretation. There exists a polynomial $Q(x)$ such that in every computation*

$$R \equiv R_{i_0+1} \xrightarrow{*} R_{i_1} \overset{Env_1}{\mapsto} R_{i_1+1} \xrightarrow{*} R_{i_2} \overset{Env_2}{\mapsto} R_{i_2+1} \dots$$

if c bounds the size of R and of the inputs $Env_1, \dots, Env_k$ for $k \geq 0$ then the size of every value computed within the instant k is bounded by $Q(c)$.

First of all we show by induction on the rank of a region that the size of every value computed in that region is polynomial in c.

If the region ρ has rank 0 the inequalities of index 2 (in the case where all auxiliary parameters are set to 0) guarantee that (i) the size of an emitted value and (ii) the size of a parameter in a recursive call to a thread identifier that may emit on the region ρ is polynomial in the parameters at the beginning of a cycle. Now, by lemma 3, the size of the parameters at the beginning of a cycle is polynomial in c. Thus from (the proof of) lemma 2, we can derive that the number of values emitted is polynomial in c. We can then conclude that all the values emitted or computed at the end of the instant by list concatenation have a size that is polynomial in c.

Next suppose the region ρ has rank greater than 0. This time the inequalities of index 2 (in the case where we restrict the auxiliary parameters to those that depend on regions of rank strictly smaller than ρ) guarantee that (i) the size of an emitted value and (ii) the size of a parameter in a recursive call to a thread identifier that may emit on the region ρ, is polynomial in the size of the parameters at the beginning of a cycle *and* the values read from regions strictly smaller than ρ. Using the fact that the composition of polynomials is again a polynomial we can appeal again to lemmas 2 and 3 to conclude that all values emitted or computed at the end of the instant by list concatenation in the region ρ have a size that is polynomial in c.

There is one situation that remains to be considered. The computation may reach a thread identifier that that does not emit any value within its current cycle. By lemma 2, it is enough to make sure that the size of its parameters is polynomial in c. This is guaranteed again by the inequalities of index 2 since by convention a region with the largest rank is in $\mathcal{W}(B)$.

Thus we have shown that the size of the values is polynomial in the size of the initial configuration and the size of the largest input. By applying again lemma 2 we can conclude that the program is feasibly reactive.

6 Conclusion

We have introduced the property of feasible reactivity in the context of a synchronous π-calculus and we have provided static conditions that enforce it. The *read-once condition* builds on the cyclic behaviour of typical synchronous applications and allows to regard each thread as a function of its parameters and of the finitely many inputs it receives within a cycle. Reactivity is obtained as usual through a *well-founded measure*. In our case, this measure is tuned so as to ensure termination in time polynomial in the size of the values. Feasible reactivity requires that we control both the number and the size of the threads. This is achieved in particular by requiring that each thread at the beginning of a cyle is *non-size increasing*. To escape certain circular situations, a final condition requires a *stratification of the signals in regions* so that, intuitively, a value emitted on a certain region can be polynomially bounded in the size of the values read in lower regions.

Various directions for further research can be mentioned. First, it is clear that an automatisation of our approach relies on the possibility of *synthesizing* quasi-interpretations. Preliminaries experiences suggest that quasi-interpretations are not too hard to find in practice (see, *e.g.*, [4]), but it remains to be seen whether this approach scales up to large programs. Second, one might wonder whether the read-once condition can be dropped. Currently, it plays an essential role in the proofs and its eventual removal seems to require new ideas on the abstraction of threads' execution. Third, our analysis is tailored towards the synchronous model and a signal based interaction mechanism. It remains to be seen whether similar analyses could be performed on different models of concurrent threads. For instance a model based on *shared references* and possibly *asynchronous* execution.

References

[1] R. Amadio. The SL synchronous language, revisited. *Journal of Logic and Algebraic Programming*, 70:121-150, 2007.

[2] R. Amadio. A synchronous π-calculus. Technical Report, Université Paris 7, Laboratoire PPS, June 2006. `http://hal.ccsd.cnrs.fr/PPS/`. To appear in *Information and Computation*.

[3] R. Amadio, G. Boudol, F. Boussinot and I. Castellani. Reactive programming, revisited. In Proc. Workshop on *Algebraic Process Calculi: the first* 25 *years and beyond*, *Electronic Notes in Theoretical Computer Science*, 162:49-60, 2006.

[4] R. Amadio. Synthesis of max-plus quasi-interpretations. In *Fundamenta Informaticae*, 65(1-2):29–60, 2005.

[5] R. Amadio, S. Dal-Zilio. Resource control for synchronous cooperative threads. In Theoret. Comp. Sci, 358:229-254, 2006.

[6] R. Amadio, F. Dabrowski. Feasible reactivity for synchronous cooperative threads. In *Proc. EXPRESS*, ENTCS, 154(3), 2006,

[7] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[8] G. Berry and G. Gonthier, The Esterel synchronous programming language. *Science of computer programming*, 19(2):87–152, 1992.

[9] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On termination methods with space bound certifications. In *Proc. Perspectives of System Informatics*, Springer LNCS 2244, 2001.

[10] F. Boussinot and R. De Simone, The SL Synchronous Language. *IEEE Trans. on Software Engineering*, 22(4):256–266, 1996.

[11] A. Cobham. The intrinsic computational difficulty of functions. In *Proc. Logic, Methodology, and Philosophy of Science II*, North Holland, 1965.

[12] M. Hofmann. The strength of non size-increasing computation. In *Proc. ACM-POPL*, 2002.

[13] N. Jones. *Computability and complexity, from a programming perspective*. MIT-Press, 1997.

[14] D. Leivant. Predicative recurrence and computational complexity i: word recurrence and poly-time. *Feasible mathematics II, Clote and Remmel (eds.)*, Birkhäuser:320–343, 1994.

[15] L. Mandel and M. Pouzet. ReactiveML, a reactive extension to ML. In *Proc. ACM Principles and Practice of Declarative Programming*, 2005.

[16] R. Milner. Communication and Concurrency. Prentice-Hall, 1989.

[17] Reactive Programming, INRIA, Mimosa Project. `http://www-sop.inria.fr/mimosa/rp`.

[18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts 1-2. *Information and Computation*, 100(1):1–77, 1992.

[19] M. Serrano, F. Boussinot, and B. Serpette. Scheme fair threads. In *Proc. ACM Principles and practice of declarative programming*, 2004.

A Proof of lemma 1

Suppose $a_{n-1}, \ldots, a_0$ are natural numbers strictly smaller than a constant c. We define

$$B_{lex}(a_{n-1}, \ldots, a_0)(c) = \Sigma_{i=0,\ldots,n-1} a_i c^i$$

which is simply the value in base c of the sequence $(a_{n-1}, \ldots, a_0)$. We also define

$$B_{mset}(a_{n-1}, \ldots, a_0)(c) = \Sigma_{i=0,\ldots,n-1} a_{\pi(i)} c^i$$

where π is a permutation over $\{0, \ldots, n-1\}$ such that $a_{\pi(0)} \leq \cdots \leq a_{\pi(n-1)}$. The permutation π is not uniquely determined but the definition of B_{mset} does not depend on its choice.

Now suppose $a_{n-1}, \ldots, a_0, b_{n-1}, \ldots, b_0$ are natural numbers strictly smaller than a constant c and note that $B_{st}(a_{n-1}, \ldots, a_0)(c) < c^n$ for $st \in \{lex, mset\}$. If $(a_{n-1}, \ldots, a_0) >_{lex} (b_{n-1}, \ldots, b_0)$ then clearly $B_{lex}(a_{n-1}, \ldots, a_0)(c) > B_{lex}(b_{n-1}, \ldots, b_0)(c)$. Therefore, the length of a decreasing sequence with respect to the lexicographic order is bounded by c^n.

On the other hand, suppose $M = \{\!|a_{n-1}, \ldots, a_0|\!\} >_{mset} \{\!|b_{n-1}, \ldots, b_0|\!\} = N$. Also assume that π, π' are permutations such that $a_{\pi(0)} \leq \cdots \leq a_{\pi(n-1)}$ and $b_{\pi'(0)} \leq \cdots \leq b_{\pi'(n-1)}$. By definition of the multi-set order, we know that there is a non-empty multi-subset of M whose largest element is, say, a which is replaced in N by another multi-set (with the same cardinality) whose largest element is strictly smaller than a. For instance, $\{\!|1, 2, 5, 5, 5, 7|\!\} >_{mset} \{\!|4, 4, 4, 4, 5, 7|\!\}$ and $\{\!|1, 2, 5, 5|\!\}$ is replaced by $\{\!|4, 4, 4, 4|\!\}$. Then for some $k \in \{0, \ldots, n-1\}$ we have: $a_{\pi(n-1)} = b_{\pi'(n-1)}, \ldots, a_{\pi(k+1)} = b_{\pi'(k+1)}, a = a_{\pi(k)} > b_{\pi'(k)}$. If follows that $B_{mset}(a_{n-1}, \ldots, a_0)(c) > B_{mset}(b_{n-1}, \ldots, b_0)(c)$ and again the length of a decreasing sequence with respect to the multi-set order is bounded by c^n.

B Abstraction

We are given a finite system of recursive equations. Our goal is to analyse the possible computations of a program whose initial shape is $R = \nu \mathbf{s}\ (A_1(\mathbf{v}_1) \mid \cdots \mid A_n(\mathbf{v}_n))$. We will assume that initially all thread identifiers are reset points, *i.e.*, $A_1, \ldots, A_n \in \mathit{Reset}$. We will proceed in two steps. First, we will *abstract* the program (the system of equations, actually) as a *term-rewriting system*. Second, we will show that the inequalities we have produced in table 1 guarantee feasible reactivity for the abstracted system and therefore for the concrete one.

B.1 Abstracting signal names

The only information we will keep of a signal name is its type $\mathit{Sig}_\rho(t)$. Thus we know its region ρ and the type of the values it may carry. Formally, we select a distinct canonical *constant*, say s, for every type $\mathit{Sig}_\rho(t)$ and replace in the program every occurrence of

a signal name of the same type with s. Following this operation, we remove all name generation instructions νs. As for the operation $[s_1 = s_2]P_1, P_2$ that compares signal names, we will simply disregard it and systematically explore the situations where one of the programs P_1 or P_2 is executed. This is like replacing a conditional $[s_1 = s_2]P_1, P_2$ with an internal choice $P_1 \oplus P_2$.

B.2 Abstracting pattern matching

Consider a pattern matching instruction $[x \trianglerighteq p]\ P_1, P_2$. As in the name comparison operation, we will systematically consider the situations where P_1 or P_2 are executed. However, in the case where the first branch P_1 is selected, we will remember that x must match the pattern p.

B.3 Abstracting the input

In section 4, we have associated a distinct label (a variable) y with every input. We rely on this variable to compute 'abstractly' beyond an input. Namely, in the input operations, say, $s^y(x).P, A(f(!^{y'}s'))$ we will consider both the possibility where a signal is received on s and the computation continues within the instant with $[y/x]P$ and the possibility that the computation suspends and resumes in the following instant with $A(f(y'))$.

B.4 Rewriting rules

We will rely on rewriting rules of the shape

$$A^+(\mathbf{p}) \to \overline{s}e \tag{3}$$

to express the situation where the thread identifier A with parameters and inputs that match the patterns $\mathbf{p}$ emits within the same instant the value resulting from the evaluation of the expression e on the signal s.

We will also rely on rewriting rules of the shape:

$$A^+(\mathbf{p}) \rightsquigarrow T \tag{4}$$

to describe the situation where the thread identifier A with parameters and inputs that match the patterns $\mathbf{p}$ evolves into a continuation T. Here, the reduction symbol $\rightsquigarrow$ can be either $\to$ or $\mapsto$ with the convention that we use $\to$ to describe a situation where the continuation T runs in the same instant and $\mapsto$ to describe a situation where the continuation T runs in the following instant.

Moreover, the continuation T can have two shapes:

- Either $B \notin \mathit{Reset}$, $T = B^+(\mathbf{e}, \mathbf{y}_B)$, and $\mathbf{p} = \mathbf{p}', \mathbf{y}_B$,
- or $B \in \mathit{Reset}$ and $T = \lambda \mathbf{y}_B.B^+(\mathbf{e}, \mathbf{y}_B)$.

Thus the rule (4) is declined into *four* cases: the continuation T can run in the same instant or not and it can be a reset point or not.

Here the notation $\mathbf{e}, \mathbf{y}_B$ (or $\mathbf{p}', \mathbf{y}_B$) should be understood with a grain of salt. We just mean that the parameters can be *partitioned* into two groups one of which corresponds to the auxiliary variables of the thread identifier B; the parameters $\mathbf{y}_B$ do not necessarily follow the others. In case $\mathbf{y}_B$ is empty, we will take the convention that $\lambda\mathbf{y}_B$ is a dummy abstraction. As usual in term-rewriting, it is assumed that the variables free in the emitted expression e or the continuation T are contained in the variables in the patterns $\mathbf{p}$ (recalling that the abstraction of a signal name is treated as a constant).

B.5 Generating the rewriting rules

Given a finite system of recursive equations, the computation of the term rewriting rules follows quite closely the generation of the inequalities described in table 1. Namely for each equation $A(\mathbf{x}) = P$ we compute the function $\mathcal{R}(P, A^+(\mathbf{x}, \mathbf{y}_A))$ which is defined on the structure of P as follows:

$$
\begin{array}{ll}
\mathcal{R}(P, A^+(\mathbf{p})) & = \mathsf{case}\ P\ \mathsf{of} \\
0 & : \emptyset \\
[x \trianglerighteq p]\ P_1, P_2 & : \mathcal{R}(P_1, A^+([p/x]\mathbf{p})) \cup \mathcal{R}(P_2, A^+(\mathbf{p})) \\
[s_1 = s_2]P_1, P_2 & : \mathcal{R}(P_1, A^+(\mathbf{p})) \cup \mathcal{R}(P_2, A^+(\mathbf{p})) \\
(P_1 \mid P_2) & : \mathcal{R}(P_1, A^+(\mathbf{p})) \cup \mathcal{R}(P_2, A^+(\mathbf{p})) \\
\nu s\ P' & : \mathcal{R}(P', A^+(\mathbf{p})) \\
\overline{s}e & : \{A^+(\mathbf{p}) \to \overline{s}e\} \\
B(\mathbf{e}) & : \begin{cases} \{A^+(\mathbf{p}) \to B^+(\mathbf{e}, \mathbf{y}_B)\} & \text{if } B \notin \mathit{Reset} \\ \{A^+(\mathbf{p}) \to \lambda\mathbf{y}_B.B^+(\mathbf{e}, \mathbf{y}_B)\} & \text{if } B \in \mathit{Reset} \end{cases} \\
s^y(x).P', B(\mathbf{r}) & : \begin{cases} \mathcal{R}([y/x]P', A^+(\mathbf{p})) \cup \{A^+(\mathbf{p}) \mapsto B^+(\overline{\mathbf{r}}, \mathbf{y}_B)\} & \text{if } B \notin \mathit{Reset} \\ \mathcal{R}([y/x]P', A^+(\mathbf{p})) \cup \{A^+(\mathbf{p}) \mapsto \lambda\mathbf{y}_B.B^+(\overline{\mathbf{r}}, \mathbf{y}_B)\} & \text{if } B \in \mathit{Reset} \end{cases}
\end{array}
$$

Here the abstracted variables $\lambda\mathbf{y}_B$ are supposed to be fresh. Also note that by the shape of the rules we can never rewrite an emission $\overline{s}e$ or an abstraction such as $\lambda\mathbf{y}_B.B^+(\mathbf{e}, \mathbf{y}_B)$ since these terms never match the left-hand side of a rule.

Example 10. *We compute the term rewriting rules associated with our running examples. For example 1, we derive:*

$$
\begin{gathered}
\mathit{Cell}^+(s, q, \ell, y) \to \mathit{Send}^+(s, q, \ell, \ell, y) \\
\mathit{Send}^+(s, q, \ell, \mathsf{cons}(s', \ell''), y) \to \mathit{Send}^+(s, q, \ell, \ell'', y) \\
\mathit{Send}^+(s, q, \ell, \mathsf{cons}(s', \ell''), y) \to \overline{s'}q \\
\mathit{Send}^+(s, q, \ell, \ell', y) \mapsto \lambda y'.\mathit{Cell}^+(s, \mathit{next}(q, y), \ell, y')
\end{gathered}
$$

For example 2, we derive:

$$\begin{array}{cc} \mathit{Server}^+(s,y) \mapsto \mathit{Handle}^+(s,y) & \mathit{Handle}^+(s,\mathsf{cons}(\mathsf{req}(s',x),\ell')) \to \mathit{Handle}^+(s,\ell') \\ \mathit{Handle}^+(s,\mathsf{cons}(\mathsf{req}(s',x),\ell')) \to \overline{s'}f(x) & \mathit{Handle}^+(s,\ell) \mapsto \lambda y.\mathit{Server}^+(s,y) \end{array}$$

Finally, for example 3, we derive:

$$\begin{array}{cc} A^+(s,y) \mapsto B^+(s,y) & B^+(s,\mathsf{cons}(n,\ell')) \to \overline{s}n \\ B^+(s,\mathsf{cons}(n,\ell')) \to B^+(s,\ell') & B^+(s,\ell) \mapsto \lambda y.A^+(s,y) \\ C^+(s) \to \overline{s}n & C^+(s) \mapsto \lambda().C^+(s) \end{array}$$

Remark 1. *The reader might have noticed that the rewriting rules and the inequalities we have produced do not keep track of events that can happen in* parallel *like "emitting two signals and calling another thread". This information can be neglected because we have assumed we are handling* finite control *programs. In such programs a call to an identifier A may generate at most one call to another thread identifier (either in the current instant or in the following one) plus a number of emissions that is bounded by a constant that depends on the size of the program only. Alternatively, we could have considered rewriting rules such as:*

$$A^+(\mathbf{p}) \to \overline{s_1}e_1 \parallel \overline{s_2}e_2 \parallel B^+(\mathbf{e},\mathbf{y}_B)$$

where the right hand side carries a composition operator $\parallel$ to express the parallelism of the events. We note that this approach may produce exponentially more rules than the previous one because one needs to distribute the parallel composition through the non-determinism.

C Analysis

We proceed to an analysis of the abstracted system, *i.e.*, of the term rewriting system. Table 2 summarizes the inequalities that are associated with each kind of term rewriting rule.

A term rewriting rule describes a family of *ground* rewriting rules which is obtained by replacing the variables with ground substitutions σ and by evaluating the ground expressions according to the evaluation axioms. We write

$$A^+(\mathbf{v}) \overset{R1}{\to} \overline{s}v$$

if there is a term rewriting rule $A^+(\mathbf{p}) \to \overline{s}e$ and a ground substitution σ such that $\sigma\mathbf{p} = \mathbf{v}$ and $\sigma e \Downarrow v$. In a similar way, we write

$$A^+(\mathbf{v},\mathbf{u}) \overset{R2}{\to} B^+(\mathbf{v}',\mathbf{u}) \qquad (\text{or} \qquad A^+(\mathbf{v},\mathbf{u}) \overset{R4}{\mapsto} B^+(\mathbf{v}',\mathbf{u})\ )$$

if there is a term rewriting rule $A^+(\mathbf{p},\mathbf{y}_B) \to B^+(\mathbf{e},\mathbf{y}_B)$ (or $A^+(\mathbf{p},\mathbf{y}_B) \mapsto B^+(\overline{\mathbf{r}},\mathbf{y}_B)$) and a ground substitution σ such that $\sigma\mathbf{p} = \mathbf{v}$, $\sigma\mathbf{y}_B = \mathbf{u}$, and $\sigma\mathbf{e} \Downarrow \mathbf{v}'$ (or $\sigma\overline{\mathbf{r}} \Downarrow \mathbf{v}'$). Finally, we write

$$A^+(\mathbf{v}) \overset{R3}{\to} \lambda\mathbf{y}_B.B^+(\mathbf{v}',\mathbf{y}_B) \qquad (\text{or} \qquad A^+(\mathbf{v}) \overset{R5}{\mapsto} \lambda\mathbf{y}_B.B^+(\mathbf{v}',\mathbf{y}_B)\ )$$

Rewriting Rules		Associated Inequalities
$(R1)$	$A^+(\mathbf{p}) \rightarrow \overline{s}e, s : Sig_\rho(t)$	$A^+(\mathbf{p})_{\downarrow\rho} \geq_2 e$
$(R2)$	$A^+(\mathbf{p}, \mathbf{y}_B) \rightarrow B^+(\mathbf{e}, \mathbf{y}_B)$	$\begin{cases} A^+(\mathbf{p}, \mathbf{y}_B) >_0 B^+(\mathbf{e}, \mathbf{y}_B) \text{ if } A =_F B \\ A^+(\mathbf{p}, \mathbf{0})_{I_A} \geq_1 B^+(\mathbf{e}, \mathbf{0})_{I_B} \\ A^+(\mathbf{p}, \mathbf{y}_B)_{\downarrow\rho} \geq_2 B^+(\mathbf{e}, \mathbf{y}_B)_{\downarrow\rho} \text{ if } \rho \in \mathcal{W}(B) \end{cases}$
$(R3)$	$A^+(\mathbf{p}) \rightarrow \lambda \mathbf{y}_B.B^+(\mathbf{e}, \mathbf{y}_B)$	$A^+(\mathbf{p})_{I_A} \geq_1 B^+(\mathbf{e}, \mathbf{0})$
$(R4)$	$A^+(\mathbf{p}, \mathbf{y}_B) \mapsto B^+(\overline{\mathbf{r}}, \mathbf{y}_B)$	$\begin{cases} A^+(\mathbf{p}, \mathbf{0})_{I_A} \geq_1 B^+(\overline{\mathbf{r}}, \mathbf{0})_{I_B} \\ A^+(\mathbf{p}, \mathbf{y}_B)_{\downarrow\rho} \geq_2 B^+(\overline{\mathbf{r}}, \mathbf{y}_B)_{\downarrow\rho} \text{ if } \rho \in \mathcal{W}(B) \end{cases}$
$(R5)$	$A^+(\mathbf{p}) \mapsto \lambda \mathbf{y}_B.B^+(\overline{\mathbf{r}}, \mathbf{y}_B)$	$A^+(\mathbf{p})_{I_A} \geq_1 B^+(\overline{\mathbf{r}}, \mathbf{0})$

Table 2: Inequalities associated with the term rewriting rules

if there is a term rewriting rule $A^+(\mathbf{p}) \rightarrow \lambda \mathbf{y}_B.B^+(\mathbf{e}, \mathbf{y}_B)$ (or $A^+(\mathbf{p}) \mapsto \lambda \mathbf{y}_B.B^+(\overline{\mathbf{r}}, \mathbf{y}_B)$) and a ground substitution σ such that $\sigma\mathbf{p} = \mathbf{v}$, and $\sigma\mathbf{e} \Downarrow \mathbf{v}'$ (or $\sigma\overline{\mathbf{r}} \Downarrow \mathbf{v}'$).

Consider a ground rewriting rule representing a computation step. As we have seen this rule is an instance of a term rewriting rule. In turn, we have associated a set of inequalities with every term rewriting rule. Let us now assume we have an assignment q that satisfies all generated inequalities. Table 3 spells out what this means in terms of the ground rewriting rule. To this end, we need some notation to distinguish the parameters $\mathbf{e}$ of a thread identifier A^+ (remember that a list of variables $\mathbf{y}_B$ or a list of patterns $\mathbf{p}$ is also a list of expressions and that $\overline{\mathbf{r}}$ is a list of expressions too since, by definition, the dereferenced signals are replaced by variables). We distinguish between proper parameters and auxiliary parameters (those corresponding to an input). Among the former, we distinguish those in the set I_A ($\mathbf{e}_{I_A}$) and the others ($\mathbf{e}_{\overline{I_A}}$). Among the latter, for a given region ρ, we distinguish those whose rank is smaller than ρ ($\mathbf{e}_{\downarrow\rho}$) and the others ($\mathbf{e}_{\overline{\downarrow\rho}}$). To summarise, given a list of parameters $\mathbf{e}$ and a region ρ, we can always partition it into four parts: $\mathbf{e} = \mathbf{e}_{I_A}, \mathbf{e}_{\overline{I_A}}, \mathbf{e}_{\downarrow\rho}, \mathbf{e}_{\overline{\downarrow\rho}}$.

We also notice that $A^+(\mathbf{e}, \mathbf{y}_A)_{I_A} = A^+(\mathbf{e}, \mathbf{0})_{I_A}$ since by definition all auxiliary parameters are set to $\mathsf{0}$. Moreover, if A is a reset point then $A^+(\mathbf{e}, \mathbf{0})_{I_A} = A^+(\mathbf{e}, \mathbf{0})$ since for the reset points, I_A coincides with the proper parameters. Finally, we recall that the restriction $\downarrow \rho$ acts only on the auxiliary parameters.

C.1 Proof of lemma 2

We analyse ground reductions of the shape:

$$A_1^+(\mathbf{v}_1) \stackrel{R2}{\rightarrow} \cdots \stackrel{R2}{\rightarrow} A_k^+(\mathbf{v}_k)$$

These reductions correspond to a sequence of recursive calls that happen within the same instant (and the same cycle). Suppose the maximum arity of a thread identifier A^+ in

$$(R1)\quad \frac{\begin{array}{c} A^+(\mathbf{p}) \to \overline{s}e, \quad s : Sig_\rho(t), \quad \mathbf{p} = \mathbf{p}_1, \mathbf{p}_2, \quad \mathbf{p}_2 = \mathbf{p}_{\overline{\downarrow\rho}}, \\ \sigma(\mathbf{p}_1, \mathbf{p}_2) = \mathbf{v}_1, \mathbf{v}_2, \quad \sigma e \Downarrow v \end{array}}{q \models A^+(\mathbf{v}_1, \mathbf{0}) \geq v}$$

$$(R2)\quad \frac{\begin{array}{c} A^+(\mathbf{p}, \mathbf{y}_B) \to B^+(\mathbf{e}, \mathbf{y}_B), \quad \rho \in \mathcal{W}(B), \quad \mathbf{p}, \mathbf{y}_B = \mathbf{p}_1, \ldots, \mathbf{p}_4, \mathbf{y}_5, \mathbf{y}_6, \\ \mathbf{p}_1 = \mathbf{p}_{I_A}, \quad \mathbf{p}_2 = \mathbf{p}_{\overline{I_A}}, \quad \mathbf{p}_3 = \mathbf{p}_{\downarrow\rho}, \quad \mathbf{p}_4 = \mathbf{p}_{\overline{\downarrow\rho}}, \quad \mathbf{y}_5 = (\mathbf{y}_B)_{\downarrow\rho}, \quad \mathbf{y}_6 = (\mathbf{y}_B)_{\overline{\downarrow\rho}}, \\ \mathbf{e} = \mathbf{e}_1, \mathbf{e}_2, \quad \mathbf{e}_1 = \mathbf{e}_{I_B}, \quad \mathbf{e}_2 = \mathbf{e}_{\overline{I_B}} \\ \sigma(\mathbf{p}_1, \ldots, \mathbf{p}_4, \mathbf{y}_5, \mathbf{y}_6) = \mathbf{v}_1, \ldots, \mathbf{v}_4, \mathbf{u}_5, \mathbf{u}_6 = \mathbf{v}, \mathbf{u}, \quad \sigma(\mathbf{e}_1, \mathbf{e}_2) \Downarrow (\mathbf{v'}_1, \mathbf{v'}_2) = \mathbf{v'} \end{array}}{\begin{array}{c} q \models (\mathbf{v}, \mathbf{u}) >_{st} (\mathbf{v'}, \mathbf{u}), \text{ if } A =_F B, status(A) = status(B) = st, \\ q \models A^+(\mathbf{v}_1, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}) \geq B^+(\mathbf{v'}_1, \mathbf{0}, \mathbf{0}, \mathbf{0}), \\ q \models A^+(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{0}, \mathbf{u}_5, \mathbf{0}) \geq B^+(\mathbf{v'}_1, \mathbf{v'}_2, \mathbf{u}_5, \mathbf{0}) \end{array}}$$

$$(R3)\quad \frac{\begin{array}{c} A^+(\mathbf{p}) \to \lambda \mathbf{y}_B.B^+(\mathbf{e}, \mathbf{y}_B), \quad \mathbf{p} = \mathbf{p}_1, \mathbf{p}_2, \quad \mathbf{p}_1 = \mathbf{p}_{I_A}, \\ \sigma(\mathbf{p}_1, \mathbf{p}_2) = \mathbf{v}_1, \mathbf{v}_2, \quad \sigma \mathbf{e} \Downarrow \mathbf{v'} \end{array}}{q \models A^+(\mathbf{v}_1, \mathbf{0}) \geq B^+(\mathbf{v'}, \mathbf{0})}$$

$$(R4)\quad \frac{\begin{array}{c} A^+(\mathbf{p}, \mathbf{y}_B) \mapsto B^+(\overline{\mathbf{r}}, \mathbf{y}_B), \quad \rho \in \mathcal{W}(B), \quad \mathbf{p}, \mathbf{y}_B = \mathbf{p}_1, \ldots, \mathbf{p}_4, \mathbf{y}_5, \mathbf{y}_6, \\ \mathbf{p}_1 = \mathbf{p}_{I_A}, \quad \mathbf{p}_2 = \mathbf{p}_{\overline{I_A}}, \quad \mathbf{p}_3 = \mathbf{p}_{\downarrow\rho}, \quad \mathbf{p}_4 = \mathbf{p}_{\overline{\downarrow\rho}}, \quad \mathbf{y}_5 = (\mathbf{y}_B)_{\downarrow\rho}, \quad \mathbf{y}_6 = (\mathbf{y}_B)_{\overline{\downarrow\rho}}, \\ \overline{\mathbf{r}} = \overline{\mathbf{r}}_1, \overline{\mathbf{r}}_2, \quad \overline{\mathbf{r}}_1 = \overline{\mathbf{r}}_{I_B}, \quad \overline{\mathbf{r}}_2 = \overline{\mathbf{r}}_{\overline{I_B}} \\ \sigma(\mathbf{p}_1, \ldots, \mathbf{p}_4, \mathbf{y}_5, \mathbf{y}_6) = \mathbf{v}_1, \ldots, \mathbf{v}_4, \mathbf{u}_5, \mathbf{u}_6, \quad \sigma(\overline{\mathbf{r}}_1, \overline{\mathbf{r}}_2) \Downarrow \mathbf{v'}_1, \mathbf{v'}_2 \end{array}}{\begin{array}{c} q \models A^+(\mathbf{v}_1, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}) \geq B^+(\mathbf{v'}_1, \mathbf{0}, \mathbf{0}, \mathbf{0}), \\ q \models A^+(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{0}, \mathbf{u}_5, \mathbf{0}) \geq B^+(\mathbf{v'}_1, \mathbf{v'}_2, \mathbf{u}_5, \mathbf{0}) \end{array}}$$

$$(R5)\quad \frac{\begin{array}{c} A^+(\mathbf{p}) \mapsto \lambda \mathbf{y}_B.B^+(\overline{\mathbf{r}}, \mathbf{y}_B), \quad \mathbf{p} = \mathbf{p}_1, \mathbf{p}_2, \quad \mathbf{p}_1 = \mathbf{p}_{I_A}, \\ \sigma(\mathbf{p}_1, \mathbf{p}_2) = \mathbf{v}_1, \mathbf{v}_2, \quad \sigma \overline{\mathbf{r}} \Downarrow \mathbf{v'} \end{array}}{q \models A^+(\mathbf{v}_1, \mathbf{0}) \geq B^+(\mathbf{v'}, \mathbf{0})}$$

Table 3: What the quasi-interpretation guarantees of a ground rewriting step

a given program is n. Moreover, suppose c is a bound on the size of the values $\mathbf{v}_j$ for $j = 1, \ldots, k$. Then the length k of the reduction sequence is $O(c^n)$. To see this, notice that $A_i \geq_F A_{i+1}$ for $i = 1, \ldots k-1$. The inequality $\geq_F$ can be strict at most a constant number of times that depends on the program. Thus, it suffices to prove the assertion when $A_i =_F A_{i+1}$, for $i = 1, \ldots, k-1$ knowing that the thread identifiers have the same status st and the same arity n. By the existence of a quasi interpretation q (case $(R2)$ in table 3), we have:

$$q \models \mathbf{v}_1 >_{st} \mathbf{v}_2 >_{st} \cdots >_{st} \mathbf{v}_k$$

By the properties of assignments, we know that the interpretation of a value is proportional to its size. Thus we can conclude by applying lemma 1.

C.2 Proof of lemma 3

We analyse ground reductions of the shape:

$$A_1^+(\mathbf{v}_1) \rightsquigarrow \cdots \rightsquigarrow A_n^+(\mathbf{v}_n) \rightsquigarrow \lambda \mathbf{y}_{A_{n+1}}.A_{n+1}^+(\mathbf{v}, \mathbf{y}_{A_{n+1}})$$

where $A_1 \in \mathit{Reset}$, $\rightsquigarrow \in \{\rightarrow, \mapsto\}$, and the last reduction is optional. These reductions correspond to a sequence of recursive calls that start with a reset point and continue within a cycle (but may span several instants). Optionally, these reductions may reach another reset point. Let us denote with $(\mathbf{v}_j)_{I_{A_j}}$ the parameters whose indexes correspond to I_{A_j}. Recall that if B is a reset point then I_B coincides with the proper parameters. By the cases $(R2)$ and $(R4)$ in table 3 we have:

$$q \models A_1^+((\mathbf{v}_1)_{I_{A_1}}, \mathbf{0}) \geq \cdots \geq A_n^+((\mathbf{v}_n)_{I_{A_n}}, \mathbf{0}) \ .$$

Moreover, at the last optional step, by inspection of the cases $(R2)$ and $(R4)$ in table 3, we deduce:

$$q \models A_n^+((\mathbf{v}_n)_{I_{A_n}}, \mathbf{0}) \geq A_{n+1}^+(\mathbf{v}, \mathbf{0}) \ .$$

In other terms, we know that if starting from a call $A(\mathbf{v})$ we arrive at a call $B(\mathbf{u})$ then $q \models A^+(\mathbf{v}, \mathbf{0}) \geq B^+((\mathbf{u})_{I_B}, \mathbf{0})$. In particular, we see that, up to the quasi-interpretation, the initial configuration $A(\mathbf{v})$ dominates all the following configurations at the beginning of a cycle.

C.3 Proof of lemma 4

We analyse ground reductions of the shape:

$$A_1^+(\mathbf{v}_1) \rightsquigarrow \cdots \rightsquigarrow A_n^+(\mathbf{v}_n) \overset{R1}{\rightarrow} \overline{s}v$$

where $A_1 \in \mathit{Reset}$, $\rightsquigarrow \in \{\rightarrow, \mapsto\}$, $\rho \in \mathcal{W}(A_n)$, and the last reduction $R1$ is optional with s also belonging to region ρ.

These reductions correspond to a sequence of recursive calls that start with a reset point and continue within a cycle (but may span several instants). Optionally, these reductions may reach a point where a value is emitted.

Let c be a bound on the size of the parameters at the beginning of the computation and the size of the values emitted by the environment at the beginning of an instant. Each vector $\mathbf{v}_j$ can be decomposed in $\mathbf{v'}_j, \mathbf{v''}_j$ where $\mathbf{v''}_j$ correspond to the auxiliary parameters on regions whose rank is not smaller than ρ's. Applying cases (R_1), $(R2)$ and $(R4)$ in table 3, we deduce:

$$q \models A_1^+(\mathbf{v'}_1, \mathbf{0}) \geq \cdots \geq A_n^+(\mathbf{v'}_n, \mathbf{0}) \geq v$$

Therefore we establish:

Property A The size of the emitted value v is polynomial in c and the size of the values read in regions whose rank is smaller than ρ's.

How many times can a value be emitted on a region ρ within an instant? Between two calls, a thread can only emit a number of messages which is bounded by a constant. Therefore, as in lemma 2, it is enough to focus on the length of computations that happen within an instant. We focus on ground reductions of the shape:

$$A_1^+(\mathbf{v}_1) \rightsquigarrow \cdots \rightsquigarrow A_k^+(\mathbf{v}_k) \stackrel{R2}{\rightarrow} \cdots \stackrel{R2}{\rightarrow} A_n^+(\mathbf{v}_n)$$

where $A_1 \in \mathit{Reset}$, $\rightsquigarrow \in \{\rightarrow, \mapsto\}$, $A_k =_F \cdots =_F A_n$, $st = \mathit{status}(A_k) = \cdots = \mathit{status}(A_n)$, and $\rho \in \mathcal{W}(A_j)$, for $j = k, \ldots, n$.

These reductions are a particular case of those considered above, where we suppose that after an initial sequence of recursive calls the computation reaches a series of calls among thread identifiers that can mutually call each other. Let $\mathbf{u}$ be the arguments that correspond to the auxiliary parameters of A_j^+ for $j = k, \ldots, n$. Each vector $\mathbf{v}_j$ can be decomposed in $\mathbf{v'}_j, \mathbf{u}$ for $j = 1, \ldots, n$.

Applying cases $(R2)$ and $(R4)$ in table 3, we deduce:

$$q \models A_1^+(\mathbf{v}_1)_{\downarrow\rho} \geq \cdots \geq A_k^+(\mathbf{v'}_k, (\mathbf{u})_{\downarrow\rho}, \mathbf{0})) \geq A_n^+(\mathbf{v'}_n, (\mathbf{u})_{\downarrow\rho}, \mathbf{0}))$$

Property B The parameters $\mathbf{v'}_j$ for $j = k, \ldots, n$ are polynomial in c and the size of values read in regions whose rank is smaller than ρ's.

Remember that by construction there is always a region ρ in $\mathcal{W}(A_k)$. Therefore, property B guarantees that the size of the proper parameters of a call to a thread identifier is under control.

Now, applying case $(R2)$ in table 3, we deduce:

$$q \models (\mathbf{v'}_k, \mathbf{u}) >_{st} \cdots >_{st} (\mathbf{v'}_n, \mathbf{u})$$

Because the values $\mathbf{u}$ are constant, we are forced to decrease the parameters $\mathbf{v'}_j$ with respect to the status st. By Property B, these parameters are polynomial in c and the size of the

values read in regions whose rank is smaller than ρ's. By lemma 2, we know that the length of the sequence is polynomial in the size of the largest parameter. Thus we compose the polynomials to obtain the following.

Property C The number of times a value can be emitted within an instant in a region ρ is polynomial in c and the size of values read in regions whose rank is smaller than ρ's.

It remains to analyse how in our model the size of the values *read* from a region depends on the size of the values *emitted* in that region. We have the following property.

Property D The values read from a region ρ are the concatenation of some of the values emitted in the region ρ within the same instant.

We can now proceed by induction on the rank of region ρ to show that the size of the concatenation of some of the values emitted in the region ρ is polynomial in c. At rank 0, we use directly properties A and C noticing that the concatenation of polynomially many values whose size is polynomial in c produces a value which is again polynomial in c. At rank $n+1$, we use again properties A and C and the inductive hypothesis. Obviously the *degree* of the polynomial will depend on the highest rank of a region which depends on the program only.

C.4 Proof of theorem 1

We can now conclude our proof. Since the size of the computed values is polynomial in c, we can apply lemma 2 and derive that each instant terminates in time polynomial in c. Thus the existence of a quasi-interpretation entails feasible reactivity.